Writing Inline Assembly 20

Assembly language lets you write the most efficient code possible. Using it with THINK C is a powerful combination. This chapter tells you how to use assembly language in your THINK C programs with the inline assembler. If you want to use an assembly language library in your project, see Chapter 19, "Porting Code."

What you should know

You should know the assembly language for the MC68000 family of microprocessors. Motorola publishes the *M68000 Family Programmer's Reference Manual*, which contains descriptions for the MC680x0 processors, and the MC6888x floating-point coprocessor.

Contents

Using the Inline Assembler 381
- Writing assembly language code 381
- Writing processor-specific assembly code 383
- Using C identifiers in assembly language 384
- Using floating-point literals 386
- Labels and branching 386
- Returning from a function 389
- Multiple entry points 389
- Using the Macintosh Toolbox in assembly language . . . 389
- Register usage 393

Differences from Other Assemblers 393
- Hexadecimal constants 393
- Directives 394
- Addresses 395

Tips 395
- Using constants 396
- Local storage 396
- Instruction size 396
- Multi-line assembly language macros 397

Using the Inline Assembler

THINK C has an integrated inline assembler that lets you write assembly language in your C routines. To use the inline assembler, set the Language Extensions option on the Language settings page of the **THINK C options** dialog to THINK C or THINK C + Objects.

You can use instructions for the Motorola MC68000 and MC68020 processors and for the MC68881 floating-point coprocessor. The inline assembler lets you refer to C variables and functions within assembly language routines. Your C routines can jump to labels in the assembly routines and vice versa.

Note

The inline assembler doesn't support any instructions that are unique to the MC68030 or the MC68040.

Writing assembly language code

The `asm` keyword invokes the inline assembler. The syntax for mixing assembly language statements with C code is simple:

```
asm {
   …     /* assembly instructions   */
   …     /* one per line            */
}
```

The compiler treats this construct as a C statement. It can appear anywhere a C statement can appear, which means within a function.

Use only one assembly language instruction per line. You can use semicolon-style comments, but because you're still writing in C, you can use C style comments as well. Preprocessor symbols and macros are expanded within assembly language, just as they are within C. You can use a C constant expression wherever a constant would be legal in assembly language.

The MC68881 and MC68882 are also called "floating-point units."

The inline assembler supports all standard MC68000, MC68020, and MC68881 instructions and addressing modes. THINK C does not support the assembly language extensions for the MC68851 memory-management unit (PMMU), so you cannot use the memory management instructions available on the MC68030 or the MC68040.

Note

Using only MC68000 assembly language ensures that your code will run on all Macintosh computers.

The inline assembler follows assembly language syntax conventions, with a few exceptions. The `DC` (define constant) directive, which places literal values in the code stream, is the only assembly language directive that the inline assembler recognizes. Use C to declare data, to define symbolic constants, and to import and to export symbols.

The following identifiers are reserved in an assembly block:

```
d0      d1      d2      d3
d4      d5      d6      d7
a0      a1      a2      a3
a4      a5      a6      a7
fp0     fp1     fp2     fp3
fp4     fp5     fp6     fp7
sp      pc      zpc     ccr
sr      usp     isp     msp
vbr     sfc     dfc     cacr
fpcr    fpsr    fpiar   dc
```

The assembler is not case sensitive with respect to instruction mnemonics and size specifications (`.B`, `.W`, `.S`, `.L`, `.P`, `.X`, and `.D`). Register names can be either all uppercase or all lowercase, but not mixed case. For example, the inline assembler recognizes `fp1` and `FP1`, but not `Fp1`.

When THINK C compiles a function that contains an assembly block, it automatically turns off some optimization and debugging options for that function. It returns those options to their original settings when it compiles the next function. The following options are turned off:

- Automatic register assignment
- Defer & combine stack adjusts
- Always generate stack frames
- All global optimizer options

Writing processor-specific assembly code

The inline assembler supports all standard MC68000, MC68020, and MC68881 instructions and addressing modes. However, to make sure your code runs on a certain Macintosh, you may need to specify instructions and addressing modes that are allowed in your assembly block. There are two ways to do this.

Warning

It's up to you to make sure that a specific processor is available before you allow your program to execute code for it. Use the Gestalt Manager to find out what processors a machine has.

Usually, you'll use the **THINK C options** dialog to make your selection. The options that affect assembly code are the Generate 68020 instructions and Generate 68881 instructions options in the Compiler Settings page of the **THINK C options** dialog. These options specify the type of assembly you can use in every `asm { … }` block.

For more information on the Generate 68020 instructions option, see "Compiling code for the MC68020" in Chapter 16. For more information on the Generate 68881 instructions option, see "Compiling code for the MC68881" in Chapter 16.

This table shows how to set these options for a certain type of assembly code:

If the 68020 option is	and the 68881 option is	You can write this type of assembly code…
Off	Off	MC68000 assembly
On	Off	MC68020 assembly
Off	On	MC68000 and MC68881 assembly
On	On	MC68020 and MC68881 assembly

Table 20-1 Setting options for assembly code

Note

You cannot use MC68020 instructions or addressing modes when the Generate 68020 instructions option is off.

Sometimes, you need to override the settings in the **THINK C options** dialog. For example, your program may be written for the MC68000, but if there is a MC68020 available, you want certain routines to take advantage of it. You can specify the processor that

an assembly block is for by putting the processor's name between the `asm` keyword and the opening brace. For example, this block is for the MC68000:

```
asm 68000 {
   …      ; MC68000 instructions
}
```

You can specify that a block is for more than one processor, usually a specific CPU and the FPU (MC68881). Put the processor names, separated with commas, between the `asm` keyword and the opening brace. For example, this block is for the MC68020 and the MC68881:

```
asm 68020, 68881 {
   …      ; MC68020 and MC68881 instructions
}
```

This table shows the processor names that the inline assembler recognizes and what code you can write for that chip:

With this name…	You can write this code…
`68000`	MC68000 assembly.
`68020`	MC68020 assembly.
`68030`	MC68020 assembly. PMMU assembly is *not* supported.
`68040`	MC68020 and MC68881 assembly. PMMU assembly is *not* supported.
`68881`	MC68881 assembly.
`68882`	MC68881 assembly.

Table 20-2 Processor names for assembly blocks

Note that the inline assembler doesn't support any instructions unique to the MC68030, MC68040, or MC68882. In fact, the names `68030`, `68040` and `68882` are synonyms for the other names.

Using C identifiers in assembly language

The inline assembler lets you use C identifiers directly. The base register (`A6` for locals, `A5` or `A4` for globals, `PC` for labels) is optional, but it must be correct if supplied.

If you've declared register variables, you can use their names wherever a register would be legal. The inline assembler is case sensitive with respect to C identifiers, just like the C compiler.

If you declare a variable to be stored in a register and THINK C can't store it in one, the assembler signals an error when you use the variable. For example, this code fragment might signal an error:

```
void foo (void)
{
   register long  *a, *b, *c, *d;
   …
   asm {
      move.l   (d), d0
      …
   }
}
```

You can reference fields of a `struct` directly. For example, if you have a variable of type `WindowRecord`, you can get the `refCon` field like this:

```
long myRefcon(void)
{
   extern WindowRecord myWindow;

   asm {
      move.l   myWindow.refCon,d0
   }
}
```

You can get the offset of a field of a structure in two ways: the simplest way is to use the name of the structure, followed by a dot, followed by the name of the field. For example, this function returns the `refCon` field of a `WindowRecord` that you have a pointer to:

```
long GetRefcon (WindowPtr wp)
{
   asm {
      movea.l  wp,a0
      move.l   WindowRecord.refCon(a0),d0
   }
}
```

The other way is to use the `offsetof()` macro defined in the file `stddef.h`. This function also returns the `refCon` field of a `WindowRecord` that you have a pointer to:

```
#include <stddef.h>  //for offsetof() macro

long GetRefcon2 (WindowPtr wp)
{
 asm {
  movea.l wp, a0
  move.l  offsetof(WindowRecord,refCon)(a0),d0
 }
}
```

Note

The macro `OFFSET()` defined in `asm.h` works the same way as does `offsetof()` and is available for compatibility with previous versions of THINK C.

Using floating-point literals

You may use valid C floating-point literals as immediate-mode operands to MC68881 instructions. When you need to specify an immediate operand that is accepted by MC68881 but that is not a valid C floating-point literal, enclose the operand in quotes. For example, in the following statement `"INF"` stands for the MC68881 representation of infinity:

```
fmove.x  #"INF",fp2
```

In the following statement, if you don't use the quotes, the assembler would attempt to store the number as an integer and would cause an overflow. The quotes tell the assembler to treat it as a double float:

```
fmove.x  #"123456678901234567890",fp1
```

With `.0` appended, the number is a valid C floating-point literal and doesn't need quotes:

```
fmove.x  #12345667890123456789O.0,fp1
```

Labels and branching

You can label assembly-language instructions with C or assembler labels. An assembler label consists of an at sign (@) followed by one

or more alphanumeric characters. Colons are optional following assembler labels, but must appear after C labels.

```
void foo (void)
{
   asm {
                …
@123            …  /* A legal asm label */
@2_px:          …  /* A legal asm label */
here:           …  /* A legal C label   */
   }
}
```

The scope of an assembler label is the enclosing function, not just the sequence of inline assembly in which it appears. This is also the way C labels work.

You can use these C branching statements within assembly language:

Statement	Action
`break`	Exits the surrounding loop or switch
`continue`	Skips to next iteration of the surrounding loop
`return`	Exits function
`goto label`	Same as `bra @label`

Table 20-3 C branching statements allowed in inline assembly

You can jump to C labels in assembly language from C code via `goto` statements.

You can also refer to C labels from inline assembly, whether the label appears in assembly code or in C code. The label must be preceded by `@` to indicate to the assembler that it is a label. This avoids ambiguity in statements such as `lea foo`, and `A1`.

The assembler optimizes branch instructions without size qualifications (such as `BNE`), choosing whether to generate a long or a short branch. This optimization does not apply to instructions with size qualifications, such as the `BNE.S` or `BNE.W` instructions.

The assembler also optimizes branches around C statements appearing within inline assembly. For example,

```
while (…) {
   …
   asm {
      bne   @1
      continue
@1    …
      …
   }
}
```

generates the same code as:

```
while (…) {
   …
   asm {
      beq   @cont
@1 …
      …
   }
cont:
}
```

As above, this optimization applies only to instructions without size qualifications; for example, it applies to the `BNE` instruction, but not to the `BNE.S` or `BNE.W` instructions.

If you `JSR` to a function from within assembly language, the function must be already declared. For example, to call the function `MyFun()` from assembly:

```
extern int MyFun(void);

int OtherFun(void)
{
   …
   asm {
      …
      jsr   MyFun
      …
   }
}
```

It's up to you to make sure that you pass the correct arguments for a function you call from assembly language. For more information, see "Calling C Routines" and "Calling Pascal Routines" in Chapter 19.

Returning from a function

Do not use the `RTS` instruction unless you are absolutely certain that you know what you're doing. If you want to return a value, always use a `return` statement. This way, the C stack frame will be cleaned up properly. For more information, see "Calling C Routines" and "Calling Pascal Routines" in Chapter 19.

Multiple entry points

You can create multiple entry points into a function by using labels declared as `extern`. First, declare each additional entry point to be a new function. Then, inside an inline assembly block, mark the beginning of each additional entry point with a label preceded by `extern`. The names of the label and the new function must be the same. For example, this code fragment creates an additional entry point into `foo()` called `foo1()`:

```
void foo1(void);

void foo(void)
{
   asm {
         moveq #0,d0
         bra.s @1
extern foo1:
         moveq #1,d0
   @1    …
         …
   }
   …
}
```

Warning

Use this feature carefully. It is very powerful, and improper use can have serious consequences.

Using the Macintosh Toolbox in assembly language

In your assembly language code, you can use any of the Macintosh Toolbox functions defined in the headers included with THINK C. The inline assembler is case sensitive with respect to trap names. If you like, you can precede trap names with an underscore.

Note

If you include `Traps.h` in your source file, you can't precede trap names with an underscore, because `Traps.h` uses the `#define` preprocessor directive to define all the trap names as constants.

Previous versions of THINK C have a built-in list of trap names and did not require that you include header files for the inline assembler.

To use Toolbox routines, you must either include `MacHeaders`, or the header file `asm.h` as well as the header file for the Toolbox routines you plan to use.

The inline assembler uses inline function definitions to generate instructions for Toolbox traps. For instance, two of the TextEdit routines are defined like this in `TextEdit.h`:

```
pascal void TECopy(TEHandle hTE) = 0xA9D5;
pascal Point TEGetPoint(short offset
   TEHandle hTE) = {0x3F3C,0x0008,0xA83D};
```

This example and those that follow are meant to show how the inline assembler generates code. Normally, you would not call these routines like this.

The inline assembler lets you use these two routines like this:

```
void F(void)
{
   asm {
            _TECopy
            _TEGetPoint
   }
}
```

It would generate machine code something like this:

```
DC.W     0xA9D5      ; trap for _TECopy
MOVE.W   #8, -(SP)   ; selector for
                     ;  _TEDispatch
DC.W     0xA83D      ; trap for _TEDispatch
```

This mechanism lets you treat certain Macintosh routines as if they were Toolbox traps even though they're really calls to dispatched traps or packages. The THINK C inline assembler takes care of calling such routines the right way. For instance, the inline assembler lets you treat the `SFGetFile` Toolbox routine as if it is a Toolbox trap:

```
asm {
   …
   _SFGetFile
   …
}
```

What the inline assembler actually generates is this:

```
asm {
   …
   move.w   #2,-(SP)     ; 2 is the routine
                         ; Selector for
                         ;  SFGetFile
   _Pack3                ; The trap for
   …                     ;  Std File
}
```

There are a few routines that have inline definitions that the assembler treats differently. For instance, the `MaxMem` Memory Manager routine is defined like this:

```
#pragma parameter __D0 MaxMem(__A1)
pascal Size MaxMem(Size *grow)
   = { 0xA11D, 0x2288 };
```

But when `_MaxMem` is used in the inline assembler, only the trap `0xA11D` is generated. These special routines, as well as traps for routines that can only be called with glue in C, are defined in `asm.h`.

The inline assembler lets you use the following 2-bit trap modifier flags to set bits 9–10 of certain traps. These trap modifiers are defined in `asm.h`:

Trap family	Modifier flag	Usage
Memory	SYS	Applies to system heap
	CLEAR	Zero allocated block
File & Driver	ASYNC	Asynchronous I/O
	HFS	Use HFS version of trap
	IMMED	Bypass driver queue
Strings	MARKS	Ignore diacritical marks
	CASE	Don’t ignore case
Get/Set Trap	NEWOS	OS trap, new numbering
	NEWTOOL	Toolbox trap, new numbering
Toolbox	AUTOPOP	Return to caller’s caller

Table 20-4 Trap modifiers allowed in inline assembly

This example is for illustrative purposes only. If you really wanted to allocate a cleared handle in the system heap, you'd use `NewHandleSysClear` *defined in* `Memory.h`.

Remember that these modifiers apply only to particular traps. Here's an example of a Toolbox routine call with a trap modifier:

```
Handle NewClearSysHandle(size)
{
   asm {
      move.l      size,d0
      _NewHandle  CLEAR+SYS
      move.l      a0,d0
   }
}
```

Most of the low-level File Manager routines (the PB routines) may be called synchronously or asynchronously. When you call these routines from C, you provide a Boolean argument to specify which way you want to call the routine, or you can use a special inline definition to call exactly the one you want:

```
void SamplePB (ParmBlkPtr pb, Boolean how)
{
   PBWrite(pb, how); /* Call it one way */
                     /*  or the other.  */

   PBWriteSync(pb);  /* Force it        */
                     /*  synchronous.   */
   PBWriteAsync(pb); /* Force it        */
}                    /*  asynchronous.  */
```

In the inline assembler, you can call the synchronous or asynchronous traps directly. If you don't specify which version of the trap you want, the inline assembler assumes you meant the synchronous one.

```
void SampleAsmPB (void)
{
   asm {
         _PBWriteSync   ;Use synch version
         _PBWriteAsync  ;Use asynch version
         _PBWrite       ;Same as
                        ; _PBWriteSync
         _PBWrite ASYNC ;Same as
                        ; _PBWriteAsync
   }
}
```

Register usage

You may modify the following registers in inline assembly code without saving and restoring their values:

```
A0          A1
D0          D1          D2
FP0         FP1         FP2         FP3
```

If you use any other registers, you should save and restore their values.

In general, if you need to use a register for other than scratch storage, you should declare a register variable and refer to it by name.

If intervening C code is executed between two stretches of inline assembly, you can assume that the C code preserves the values of registers `A5`, `A6`, and `A7`—as well as `A4` for drivers and code resources. All other registers may have been modified. It is safe to leave things on the stack while C code is executing, provided you clean up the stack before returning from the function.

For more information, see "Calling C Routines" and "Calling Pascal Routines" in Chapter 19.

Differences from Other Assemblers

This section describes the differences between THINK C's inline assembler and most other assemblers.

Note

The inline assembler strictly enforces the syntax for MC68020 addressing modes. The previous version of THINK C allowed you to change the ordering of some operands.

Hexadecimal constants

THINK C does not accept the syntax `$NNNN` to designate a hexadecimal constant. Use the C syntax `0xNNNN` instead.

THINK C treats hexadecimal constants as unsigned numbers. For example, `0xFF` is equivalent to `255`, `0xFFFF` is equivalent to `65535`, and neither is equivalent to `-1`.

Directives

Two or more DC.B directives in a row are treated as one DC.B directive. This means that a DC.B directive can occupy an odd number of bytes when another DC.B directive follows. For example:

```
dc.b  'a'
dc.b  'b'
dc.b  'c'
dc.b  'd'
```

is equivalent to:

```
dc.b  'a', 'b', 'c', 'd'
```

However, preceding a DC.B directive with a label will word-align it. For example, this:

```
    dc.b  'a'
@1  dc.b  'b'
@2  dc.b  'c'
@3  dc.b  'd'
```

will assemble in the same way as:

```
dc.b  'a', 0, 'b', 0, 'c', 0, 'd', 0
```

The DC.B directive accepts strings (both Pascal and C) as operands. The terminating null byte is not assembled. Continuing the above example, this:

```
dc.b  'a', 'b', 'c', 'd'
dc.b  5, 'e', 'f', 'g', 'h', 'i'
```

is equivalent to this:

```
dc.b  "abcd"
dc.b  "\pefghi"
```

If you want a string defined as a C string, you must supply the terminating null. You can do it like this:

```
dc.b  "ABCDE\0"
```

The DC.W directive can accept a function name or a label as an operand and generate the PC-relative offset of the function or label as a word. The function or label must be defined within the segment in which the directive appears.

The DC.S, DC.D, and DC.X directives accept floating-point constants. For example:

```
dc.s  3.14     ; single-precision float
dc.d  "inf"    ; double-precision float
dc.x  "10000000000"
               ; extended-precision float
```

Addresses

The difference between two addresses is not a constant expression, so instructions like this are not allowed:

```
move.w   #@2-@1,d0
```

Similarly, the inline assembler does not support the following syntax to assemble the PC-relative offset of the label @1:

```
dc.w     @1-*
```

Instead, use this:

```
dc.w     @1
```

For example, to code a dispatch table, use:

```
; d0 contains 0,1,2,…
   add.w d0,d0
   add.w @0(d0.w),d0
   jmp   @0(d0.w)
@0 dc.w  @1       ;  case 0
   dc.w  @2       ;  case 1
   …              ;  …
```

Tips

Inline assembly can be tricky if you are not familiar with assembly language. It can be especially dangerous if you're used to thinking in terms of high-level languages. The following problems are not specific to THINK C; they are common to assembly language in general.

Using constants

Don't forget the # sign when using an immediate constant. Otherwise, the result will be *very* different than what you intended.

```
extern short MemErr : 0x220;
                          // declare MemErr
                          // low mem global

   asm {
   move.w   0x220,D0     ;Moves contents of
                          ; location 0x220
                          ; (MemErr) into D0
   move.w   MemErr, D0   ;Same thing, but
                          ; symbolically
   move.w   #0x220, D0   ;Moves the value
                          ; 0x220into D0
   move.w   5, D0        ;NO: On an MC68000
                          ; this causes an
                          ; odd address
                          ; error.
                          ; On a MC68020 this
                          ; moves what's in
                          ; location 5.
   move.w   #5, D0       ;OK
}
```

Local storage

Use C variables to declare local storage. If you use the directive `DC` instead, storage will be allocated in your code segment. Most of the time, this is not what you want.

Instruction size

Be sure to use the right sized instruction to refer to variables. For example:

```
function()
{
   short GetsTrashed;
   short v;

   asm{
      move.l   #3,v      ;NO: Overwrites
                         ; GetsTrashed
      move.w   #3,v      ;OK: Word size
                         ; matches int.
   }
}
```

Multi-line assembly language macros

You need to write multi-line assembly language macros differently than you did with THINK C 4.0 and earlier versions. For example, this multi-line macro now causes an error in THINK C:

```
#define LongMult(r, s1, s2) \
asm {  \
   move.w  s1, d0  \
   muls  s2, d0  \
   move.l  d0, r \
}

void foo(void)
{
   int a, b;
   long d;

   LongMult(d, a, b)    // Syntax error
   . . .
}
```

Multi-line assembler macros cause errors because the THINK C preprocessor expands the macro as one, long line, as shown:

```
asm { move.w s1, d0  muls s2, do  move.l d0,r }
```

The inline assembler doesn't recognize this expansion as valid assembly language.

There are two ways to correct this problem. You can define the macro as follows:

```
#define LongMult(r, s1, s2) \
asm { move.w  s1, d0 } \
asm { muls  s2, d0  } \
asm { move.l  d0, r }
```

Or you can define `LongMult` as just the assembly language statements, without enclosing them in `asm { . . . }`. For example:

```
#define LongMult(r, s1, s2) \
move.w  s1, d0  \
muls  s2, d0  \
move.l  d0, r
```

And then use the macro like this:

```
void foo(void)
{
   int a, b;
   long d;

   asm {
      LongMult(d, a, b)
   }
   . . .
}
```